Eiffel vs C++ : One language designer's view

by Bertrand Meyer

.. Eiffel is a language and environment designed with a precise charter (enabling the production of very high quality software by professional programmers)..

I would like to start a discussion of Eiffel vs. C++. [...] About the only difference that I can discern is the fact that Bertrand Meyer posts more often to comp.lang.eiffel than Bjarne Stroustrup posts to comp.lang.c++. ;-)

David Geary

Eiffel vs C++

Eiffel - One language designer's view

Table of Contents

1. Introduction

2. General Observations

3. Technical Differences
   3.1 Software structure
   3.2 Assertions
   3.3 Exceptions
   3.4 Global variables

http://www.elj.com/eiffel/bm/eiffelvscpp89/
3.5 Genericity
3.6 Dynamic binding
3.7 In-line expansion
3.8 Operator overloading
3.9 Consistency of the type system
3.10 Type checking
3.11 Friend functions
3.12 Deferred classes
3.13 Multiple inheritance
3.14 Renaming
3.15 Garbage collection
3.16 Automatic recompilation
3.17 Pointer arithmetic etc.
3.18 Compatibility with C software
3.19 Simplicity and ease of learning

4 Conceptual Differences

1. Introduction

Since we often get the question of Eiffel versus C++, Mr. Geary's initiative provides a welcome opportunity to clarify what I see as the major differences. I am the main designer of Eiffel, so the following discussion cannot claim to be that of an independent observer, although it is certainly honest, and technically correct to the best of my knowledge. In all likelihood, others will contribute different viewpoints.

I certainly cannot claim to be a C++ expert. If I misrepresent C++ in any way, it will be by mistake, not design. Should this occur, someone will certainly correct the mistake, either by writing to me (in which case I will post the correction if it does turn out that I have written something wrong) or by posting directly.

This response has three parts:

- general observations;
- technical differences;
- conceptual differences.

I apologize for its length; there is really a lot to say. Also, there is little new in the material below, much of which may be found in previous publications about Eiffel; none of my previous publications, however, has studied Eiffel from the point of view of a comparison with C++.

Although I believe that the Eiffel environment and tools and the standard Eiffel libraries provide some of the major arguments for Eiffel, I will (with two exceptions explained below) limit the discussion to the languages proper, so as to abide by Mr. Geary's request:

I am aware of the fact that Eiffel has a robust library of base classes, but I would like to see a discussion of language features

2. General Observations

Eiffel was designed in 1985, initially not as a language for the rest of the world, but as an internal tool for a development that Interactive Software Engineering started at that time. We would have preferred to use an existing language and environment, but an examination of possibilities, including C++, quickly showed that nothing was even close to the minimum required for developing software according to the modern principles that I (then still a professor at the University of California, Santa Barbara) was teaching to my students. I did not want to develop a split personality or to fall into the do as I say, not as I do syndrome.

I am mentioning this because in a way we had an unfair advantage: when we started, C++ existed. Although we certainly did not imitate C++, its very existence was helpful because it showed clearly what we did not
want Eiffel to be. In a similar vein, we also looked at Ada, although in this case the ratio of positive to negative influences was higher. Even though Ada is not an object-oriented language, its syntax conventions and its handling of elementary constructs (expressions, control structures etc.) definitely influenced the corresponding Eiffel constructs.

(By the way, only three languages did exert a strong semantic influence on Eiffel. One was Simula 67; the other two were not programming languages but specification languages: Abrial's Z, in whose design I was somewhat involved around 1978–79, and my own M, a successor to Z and to this day an unfinished design. A less important influence was Alphard.)

Finally I do not think that in the long term Eiffel is really competing with C++. They have almost nothing in common in their aim and spirit, as explained in the final section of this message. In my undoubtedly biased view, the real competitor to Eiffel is Ada. (Competitor for me is a positive word. When you acknowledge a design as a competitor to yours, it implies respect.) Ada, as already mentioned, is not truly object-oriented, but its official charter (reliability, reusability, professional high-quality software development etc.) is very similar, on paper at least, to that of Eiffel.

3. Technical Differences

3.1 Software structure

Eiffel software is organized in autonomous software units (classes), meant to be compiled separately. There is no main program. This is what I believe should be the case in object-oriented programming. In contrast, I understand that C++ still follows the traditional C model. Quoting from Dr. Bjarne Stroustrup's *The C++ Programming Language* (Addison-Wesley, 1986), which seems to be the major reference on C++, page 22, lines 13-14: A C++ program typically consists of many source files, each containing a sequence of declarations of types, functions, variables, and constants. This is very far from the object-oriented model of software decomposition. Furthermore, reports from actual users of C++ seem to indicate a heavy use of include files, a technique which I don't fully understand in the C++ context, and which has no equivalent in Eiffel.

3.2 Assertions

A fundamental property of Eiffel software is that it may be equipped with assertions. Assertions are elements of formal specification that serve to characterize the semantics of classes and their routines independently of their implementation. Assertions include in particular routine preconditions (which must be satisfied when a routine is called), routine postconditions (ensured by the routine on exit) and class invariants (global consistency conditions applying to every instance of a class).

Assertions are essential for documenting components. As a matter of fact, I do not understand how one can talk about the very idea of reusable software components without assertions. Using a hardware analogy, a software component without assertions is similar to, say, an amplifier without precondition (the acceptable input voltage), postcondition (the gain, expressed as acceptable ratio of output to input) and invariant (including for example the temperature limits expected and maintained by the amplifier). Yet of widely available programming languages, only Eiffel has these notions.

(A system that does have assertions, and in fact ones that are more sophisticated than Eiffel's current ones, is David Luckham's Anna system, developed at Stanford on top of Ada. As far as I know, however, this is not a deliverable product.)

Beyond their documentation uses, assertions, which optionally may be monitored at run-time, provide a remarkable debugging and testing aid. At the recent Eiffel conference in Paris, one user organization (*Cognos Inc.*) reported that they no longer perform traditional unit testing, having replaced it by assertion monitoring.

3.3 Exceptions

Eiffel has exception handling. Its exception mechanism is original and I believe it is one of our major
contributions, based on the theory of Programming by Contract. As far as I know, there is no exception mechanism in C++. I believe that one cannot write serious software without having a way to recover cleanly from unexpected cases.

3.4 Global variables

Consistent with the absence of main program is the absence of global variables. Global variables are well known to be detrimental to modularity and more generally to quality. The Eiffel technique of once routines is used to ensure disciplined sharing between classes when needed. (See my column in the Journal of Object-Oriented Programming, vol. 1, no. 3, pages 73-77, "Bidding Farewell toGlobals"). In contrast, C++ seems to support global variables in the C style.

3.5 Genericity

Eiffel classes may be generic, i.e. parameterized by types, as in LIST [T]. Here actual uses of the class may use any type (class) as actual generic parameter, as in

    my_list: LIST [TEXT_LINE]

The genericity may be constrained, as in MATRIX [T -> NUMERIC], which specifies that actual generic parameters must be descendants, in the sense of inheritance, of class NUMERIC (equipped with the operations +, -, * etc.). Descendants of NUMERIC include (in version 2.2) predefined types such as INTEGER and REAL. The operations of NUMERIC are available, within the class, on any variable of type T - so that it can define, for example, routines for adding and multiplying matrices. Note that in this example MATRIX itself may inherit from NUMERIC.

Nothing of the sort exists in C++. This means that generic structures must be simulated by forcing type conversions, or casts, using low-level C techniques. This defeats any attempt at static typing.

A paper was published not long ago to describe a proposal for class parameterization in C++.

(Although the paper was published in a refereed journal, it regrettably did not mention any of the two object-oriented languages that offer such a facility: Eiffel and Trellis-Owl, the latter designed by Craig Schaffert and others from DEC).

Since by all reliable accounts the inclusion of such a facility in any form accessible to C++ users is several years away, it cannot be considered in any serious discussion. On that kind of time scale one can promise anything.

3.6 Dynamic binding

Dynamic binding is the default mechanism for routine calls in Eiffel (achieved without any undue effect on performance). The default policy in C++ is static binding; dynamic binding is only applied to routines declared as virtual. This may look like an acceptable requirement to impose on programmers but I believe it is not. The whole idea of inheritance is that you may reuse a class later on by writing a descendant and adapting it to new uses by overriding some of the routines of the original - within the original semantic constraints, as defined by assertions. This should be done without impacting the original, which may be used by many other client classes. (These concepts are explained in my book Object-Oriented Software Construction, Prentice-Hall, 1988, as the Open-Closed Principle, section 2.3.) In such a case the designer of the original routine may have had no inkling whatsoever that the routine would ever be redefined and subjected to dynamic binding.

This is incompatible with the requirement that the original designer should have declared the routine as virtual in the first place.

Instead of forcing the programmer to take care of low-level optimizations, the Eiffel approach makes the compiler responsible for exploiting the performance of static over dynamic binding. The optimizer, working on a set of classes, generates code that applies static binding to any routine which warrants it (because it is never redefined).
Performing tedious and potentially dangerous optimizations in a safe way should be the role of computers, not humans.

3.7 In-line expansion

In C++ as in Ada a routine may be declared as **in-line**, meaning that calls will be expanded in-line to gain performance. No such mechanism is available in Eiffel. Contrary to what one might think at first sight, I believe this to be a serious advantage for Eiffel. As soon as a routine is declared as in-line, its usefulness is severely limited because it no longer is a normal routine that can be redefined and subjected to dynamic binding. The discussion of the previous paragraph applies even more strongly.

In Eiffel, once again, the corresponding optimizations are performed by the compiler, not by the human user. The optimizer will automatically expand certain routines in-line based on systematic criteria beyond programmer control. One of the criteria is of course that the routine not be subject to redefinition and dynamic binding; the number of calls in the code is another.

Again, this seems the safe and efficient approach. Computers can perform this kind of task both more efficiently and more safely than humans.

3.8 Operator overloading

The term **operator overloading** is not entirely adequate since the issue is whether functions may be assigned names that will be used in prefix or infix form in calling expressions. This is a syntactic, not a semantic issue; the more important form of overloading, the semantic one, is provided in the object-oriented context by redefinition and dynamic binding.

C++ offers the possibility of using an operator (from a set of predefined ones) as function name; a similar possibility is offered in Eiffel 2.2, although it was not present in earlier releases. So the two languages are now indeed comparable in this respect.

3.9 Consistency of the type system

Beginning with version 2.2, Eiffel has a fully consistent type system in which every type, including basic types such as **INTEGER**, **REAL** and so on, is defined by a class (using the multiple inheritance mechanism). This was made possible by the introduction of the notion of expanded class, of special **BITS M** classes (whose instances are bit strings of length M), and of infix/prefix operators as discussed above. This is achieved without any effect on the efficiency of dealing with simple values such as integers, characters and the like. The advantage is mainly a conceptual one - being able to work with a single set of concepts admitting few special cases.

There doesn't seem to be anything similar in C++, which uses the C types as basis.

3.10 Type checking

Because of the absence of genericity and the presence of the full C type system with its casts and other unsafe mechanisms, C++ cannot be reasonably be called a statically typed language. In contrast, Eiffel was designed as fully typed.

The present Eiffel compiler misses a small number of type violations (arising in particular from cases in which polymorphism enables a client to evade an export or redefinition constraint). These cases seldom arise in practice, which is not an excuse for not handling them properly. Even with the current implementation, however, Eiffel is incomparably more type-safe than C++ because of the presence of genericity, of the strict enforcement of type checks in assignments, and of the absence of any unsafe casts or conversions.

3.11 Friend functions
C++ has a notion of friend function which, as I understand it, makes it possible to define routines outside of the object-oriented framework. There is nothing equivalent in Eiffel. This facility is not missed; I would see its introduction as a dangerous violation of the object-oriented principles.

3.12 Deferred classes

An extremely important notion in Eiffel is that of deferred class, which describes a non-fully-implemented abstraction. Deferred classes are used to capture commonalities and are central to the object-oriented approach. Two aspects are particularly important: the ability to define a partially deferred class, which contains both implemented and non-implemented routines; and the ability to attach assertions to a deferred class and its deferred routines, and thus to specify the behavior of yet to be implemented software.

C++ as described in published references does not appear to support a similar notion. I have heard, however, that the forthcoming version of C++ has a notion of abstract class, which is meant to play the same role. Perhaps someone will describe this facility in detail so that readers can judge.

3.13 Multiple inheritance

Multiple inheritance is fundamental in the Eiffel approach. We made every effort to handle it in a very clean way; name clashes, in particular, are treated in what I believe is the right way.

*(More precisely, I do not know of any satisfactory solution in any other language. This is a strong statement, and proponents of other languages are welcome to respond to the challenge.)*

The published references on C++ systems exclude multiple inheritance which, however, is rumored to be imminent. I must confess this is one aspect on which we, at Interactive Software Engineering, are rather touchy. The first time I personally heard about the *imminence* of multiple inheritance in C++ was November of 1986. Since then, that is to say for two years and a half, we have essentially been unable to use the presence of multiple inheritance as an argument for Eiffel - so successful have others been in persuading almost everyone that multiple inheritance in C++ was around the corner. Here I would like to appeal to developers of software tools and suggest a universal ethical rule: whenever you refer to a feature that is not yet available in the released distribution of your product, mention it unambiguously in every relevant publication, together with an estimated date of availability (*to the public, not internally*). The name for such a policy is simple: honesty.

This being said, it is indeed possible that multiple inheritance will become available in C++ during my lifetime. It is not clear from what I have read and heard that the non-trivial problems of multiple inheritance, such as name clashes, have been properly addressed. (If I am wrong on this, please enlighten me.)

3.14 Renaming

Eiffel offers a powerful technique in connection with inheritance: renaming. A class can rename inherited routines and attributes (i.e. methods and attribute variables for those who prefer such terms). This is used for removing name clashes in multiple inheritance and also, perhaps even more importantly, to provide locally adapted terminology when you inherit the right features but under the wrong names. As discussed in my OOSC book referenced above (section 10.4.7) and in a JOOP column (Vol. 1, no. 4, pages 48-53), this is essential if inheritance is to provide support for reusability in a practical industrial context.

3.15 Garbage collection

This item and the next violate Mr. Geary's request to limit the discussion to language features. I have included them anyhow because, even though they are environment rather than language features, they are made possible or next-to-impossible by the language design.

To write serious object-oriented software, which at run time will inevitably generate many objects, some of which may become useless, one needs a good garbage collector. This is the case in Eiffel (which uses an incremental, parallel scheme so as not to impair performance). As far as I know, C++ systems do not support
garbage collection, which would be extremely difficult if not impossible to implement because of the presence of C types and mechanisms.

3.16 Automatic recompilation

One of the most important practical aspects of Eiffel is the automatic compilation mechanism, based on automatic analysis of inter-class dependencies (multiple inheritance and client). This removes the need for make files and include files. Although I recall some seemingly interminable notes on the feasibility (or lack thereof) of a similar mechanism in comp.lang.c++, I don't know of any implemented mechanism for C++. Again, this seems due to the very design of the language; and again, the difference seems to result from irreconcilable views of what should be done by computers and what should be done by humans. The Eiffel view is that error-prone and tedious management tasks should be handled by tools, and that programmers should concentrate on solving programming problems.

3.17 Pointer arithmetic etc.

One of my major objections to C++ stems from what that language has rather than what it has not. Because C++ retains almost total compatibility with C, it keeps all its low-level and dangerous features. The design of C dates back to the late sixties and is obsolete by modern software engineering standards. Compatibility with C means that in C++ you still have pointers, type casts, pointer arithmetic, function pointers, malloc, free, bizarre operator precedence (the famous asterisk/parenthesis bugs), weak type checking and so on.

I strongly disagree with this approach if the goal is to obtain software quality. Take pointer arithmetic, for example. I would contend that you can have quality software, or you can have pointer arithmetic; but you cannot have both at the same time.

In Eiffel, the choice has been made. None of these low-level features are present (as John Sarkela pointed out in a previous message); needless to say, they are not missed.

3.18 Compatibility with C software

C++ is obviously very compatible with C. But in Eiffel too you can easily communicate with C software; both call-in and call-out are provided. This makes it possible to reuse existing software easily. The need for such communication was not, however, deemed to be a good argument for impairing the consistency of the language itself.

3.19 Simplicity and ease of learning

Much of the plea for C++ is based on the observation that it provides an easy transition from C, which (for better or worse) is the language many programmers know nowadays. Using Dr. Brad Cox's expression (meant for Objective-C), this supports an evolutionary approach.

I can certainly respect this view and its appeal to software managers in industry. But I believe that by considering it more closely one will find it short-sighted and ill-founded.

Learning a new language such as Eiffel is nothing for a competent programmer. For Eiffel, which is small and simple, the learning process typically lasts a few days at most. Nobody has ever told us that Eiffel was difficult to learn. (If you read this, have tried to learn Eiffel, and found otherwise, please respond!) I believe that the process of going to Eiffel is in fact much smoother, as you don't have to use a confusing mix of old and new concepts. In a language that you master totally, you feel confident and you can concentrate on your job rather than on the language intricacies.

Also, the brief initial shock produced by the realization that you cannot easily write your programs in a traditional way any more is, in the experience reported by Eiffel users, highly salutary.

4. Conceptual Differences
The considerable differences listed above more than offset, in my mind, any similarity that may seem to exist between Eiffel and C++. Beyond these individual technical differences, the contrast between the two languages is deep and conceptual. Eiffel is a new language and environment designed with a precise charter (enabling the production of very high quality software by professional programmers). C++, as I see it, is an attempt at a more modern version of C. (Dr. Stroustrup's book, in the historical note on page 5 of his book, writes that the difference between C and C++ is primarily in the degree of emphasis on types and structures).

There are undoubtedly arguments for both approaches. Obviously, I believe that arguments for the first are much stronger.

---

The C++ Critique (Version 3) by Ian Joyner (Oct '96)

including comparisons to Java and Eiffel.”

(Ed: Banner provided as an OO Community Service.)